AFRL-IF-RS-TR-1999-23
Final Technical Report
February 1999

HYPERVISORS FOR SECURITY AND ROBUSTNESS

Secure Computing Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E286

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

AFRL-IF-RS-TR-1999-23 has been reviewed and is approved for publication.

APPROVED: Dwayne P. Allain

DWAYNE P. ALLAIN
Project Engineer

FOR THE DIRECTOR: [illegible]

WARREN H. DEBANY JR., Technical Advisor
Information Grid Division
Information Directorate

HYPERVISORS FOR SECURITY AND ROBUSTNESS

Terrence Mitchem
Kent Larson
Raymond Lu
Brian Loe
Dick O'Brien

Contractor: Secure Computing Corporation
Contract Number: F30602-96-C-0338
Effective Date of Contract: 29 August 1996
Contract Expiration Date: 29 March 1998
Short Title of Work: Kernel Hypervisors

Period of Work Covered: Aug 96 - Mar 98

Principal Investigator: Dick O'Brien
Phone: (612) 628-2765
AFRL Project Engineer:
Phone: (315) 330-7796

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Dwayne P. Allain, AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1999	3. REPORT TYPE AND DATES COVERED Final Aug 96 - Mar 98

4. TITLE AND SUBTITLE

HYPERVISORS FOR SECURITY AND ROBUSTNESS

5. FUNDING NUMBERS

C - F30602-96-C-0338
PE - 62301E
PR - E017
TA - 01
WU - 10

6. AUTHOR(S)

Terrence Mitchum, Kent Larson, Raymond Lu, Brian Loe, and Dick O'Brien

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Secure Computing Corporation
2675 Long Lake Road
Roseville MN 55113

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-23

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Dwayne P. Allain/IFGB/(315) 330-2663

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This is the final report for the Hypervisors for Security and Robustness (Kernel Hypervisors) program. It contains a description of the kernel hypervisor approach that was developed on the program for selectively controlling COTS components to provide robustness and security. Using the concept of a loadable module, kernel hypervisors were implemented on a Linux kernel. These kernel hypervisors provide unbypassable security wrappers for application specific security requirements and can also be used to provide replication services. Kernel hypervisors have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services for limiting possible compromises.

This report also includes a summary of the results of the performance testing and composability analysis that was done on the program. It concludes with a discussion of lessons learned and open issues.

14. SUBJECT TERMS

Wrappers, Security Wrappers, Kernel Loadable Modules, Computer Security, Application Security, Browser Security, Linux Security, Hypervisors

15. NUMBER OF PAGES

56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Table of Figures

Abstract

This is the final report for the Hypervisors for Security and Robustness (Kernel Hypervisors) program. It contains a description of the kernel hypervisor approach that was developed on the program for selectively controlling COTS components to provide robustness and security. Using the concept of a loadable module, kernel hypervisors were implemented on a Linux kernel. These kernel hypervisors provide unbypassable security wrappers for application specific security requirements and can also be used to provide replication services. Kernel hypervisors have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services for limiting possible compromises.

This report also includes a summary of the results of the performance testing and composability analysis that was done on the program. It concludes with a discussion of lessons learned and open issues.

Keywords: wrappers, security wrappers, kernel loadable modules, computer security, application security, browser security, Linux security, hypervisors

1. Overview

This document is the final report for the Hypervisors for Security and Robustness program sponsored by DARPA's Information Technology Office. It describes the approach, called kernel hypervisors, developed on the program for selectively wrapping COTS components to provide robustness and security, and summarizes the major achievements of the program.

1.1 Introduction

A hypervisor is a layer of software, normally operating directly on a hardware platform, that implements the same instruction set as that hardware. They have traditionally been used to implement virtual machines[1]. Kernel hypervisors are a similar concept, but are implemented on top of an operating system kernel rather than on top of the hardware. These kernel hypervisors provide a set of "virtual" system calls for selected system components.

Kernel hypervisors are loadable kernel modules that intercept system calls to perform pre-call and post-call processing. They can be used to provide an additional layer of fine-grained security control or to provide replication support. Their key features are that they can be set up to be unbypassable, since they are in the kernel, and they are easy to install, requiring no modification to the kernel or to the COTS applications that they are monitoring.

Kernel hypervisors have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services for limiting possible compromises.

1.2 Accomplishments

On this program, the following specific accomplishments were achieved.

- A framework was developed based on a master kernel hypervisor, whose job is to coordinate installation and removal of individual client kernel hypervisors and to provide a means for management of these clients. The framework allows client kernel hypervisors to be stacked so that a variety of application specific policies can be implemented, each by means of its own kernel hypervisor.
- A variety of specific client hypervisors (also called kernel loadable wrappers) were developed to test and demonstrate the feasibility of the kernel hypervisor concept Specific wrappers developed were:
    1. Netscape browser wrapper: to limit the damage that could be done when a user, browsing on the Internet, downloaded and executed malicious active content.
    2. Apache web server wrapper: to protect the server's data files from unauthorized modification and to limit the damage that could be done if

the server was taken over by a malicious user.

3. Replication wrapper: to automatically replicate files as they are being modified in a manner transparent to the applications performing the modifications.

The Netscape browser and Apache web server client hypervisors are actually generic wrappers that can be used to isolate an application and its data files in a separate domain so that the application is protected from other components of the system and vice versa.

- A kernel hypervisor management component was developed that provides the ability to dynamically reconfigure the policies that the various client hypervisors enforce.
- Performance testing was done to measure the impact of the various client hypervisors on the performance of their associated applications.
- A composability analysis was performed to analyze the security implications of stacking client hypervisors.
- A conference paper[2] describing the work was presented at the Annual Computer Security Applications Conference 97 (ACSAC97) and appears in the proceedings of that conference.
- The source code for the kernel hypervisor components developed on this program has been made available on the web for other researchers. All code was developed for the Linux operating system.

1.3 Document Organization

The remainder of this document is organized as follows.

- Section 2 describes the kernel hypervisor architecture, presents a high level view of the system components that were developed and discusses some of the possible uses of kernel hypervisors.
- Section 3 presents a summary of the results of the performance testing.
- Section 4 describes the composition analysis that was performed on the system.
- Section 5 documents some of the lessons learned on the project and outlines possible future research directions.
- The References section includes a list of cited and related documentation.
- The Appendix contains the User Guide.

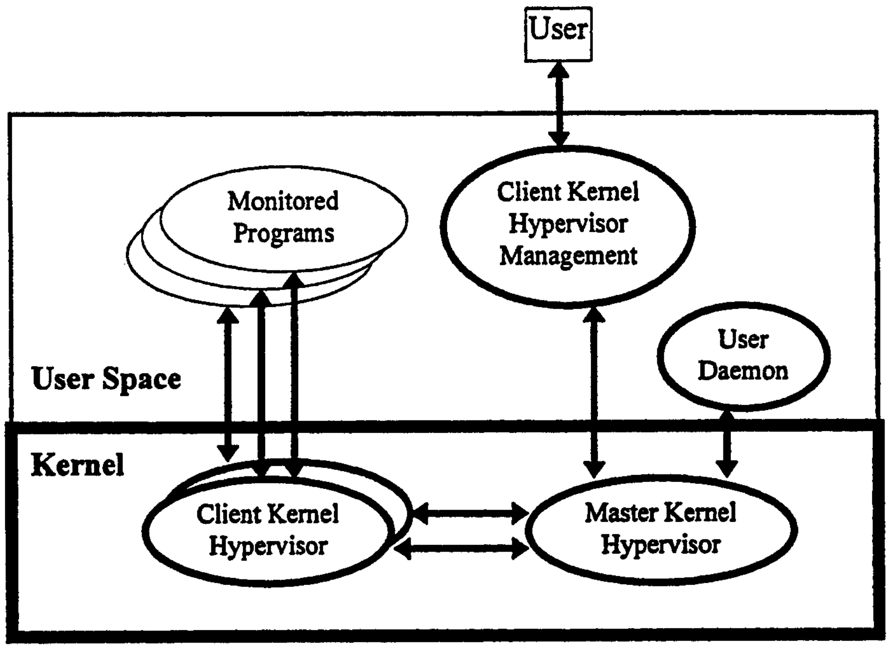
2. Kernel Hypervisor Approach

Section 2.1 presents an overview of the kernel hypervisor architecture that was developed on the program and includes a brief discussion of the benefits of the approach and how it relates to other research work. Section 2.2 then provides more details of the specific system components that were developed. More details of the implemented system can be found in the Design Report [3]. Section 2.3 documents some of the possible applications of the kernel hypervisor technology.

2.1 Architecture Overview

The kernel hypervisor architecture is illustrated in Figure 1.

Figure 1. System Components



- Master Kernel Hypervisor provides communications and control
- Client Kernel Hypervisors provide application specific monitoring
- Client Kernel Hypervisor Management provides the user interface for configuration of client hypervisors

There are three main components:

- The master kernel hypervisor: manages the individual client kernel hypervisors that are currently loaded and provides a control facility allowing users to monitor and configure these client hypervisors.
- Specific client kernel hypervisors: provide application specific policy decision making and enforcement. Each client kernel hypervisor can wrap

one or many applications. User daemons that run in user space can also be developed to allow the client hypervisors to initiate actions in user space.

- Client hypervisor management module: provides an interface for communicating with and configuring the client kernel hypervisors from user space.

Kernel hypervisors are distinguished by the fact that they consist of *loadable kernel code* that is used to wrap specific applications. Other approaches have been used to provide wrappers for additional security or robustness including:

- Traditional hypervisors that replace the standard hardware interface with an interface that provides additional capabilities. This approach was used by Bressoud and Schneider[1] for building a replication hypervisor that intercepts, buffers, and distributes signals from outside the system.
- Special libraries that include security functionality and that are linked with an application before it is run. This is the approach taken by SOCKS[4] where the client links in the SOCKS client library. SOCKS is intended for use with client/server TCP/IP applications, but the concept of linking in special libraries can be used for any type of application.
- Wrappers that make use of an operating system's debug functionality. This is the approach taken by the Berkeley group[5].

Our approach is most like the last, but it differs in that we place our code directly in the kernel and do not use the system debug functionality. Loadable kernel hypervisors have a number of benefits.

- They are unbypassable. Since the wrapper is implemented within the operating system kernel, malicious code cannot avoid the wrapper by making direct calls to the operating system as could be done with code library wrappers. Any such calls will be intercepted and monitored.
- They do not require kernel modifications. All kernel hypervisor code is implemented as modules that are loadable within the kernel while the system is running. There are no changes to kernel source code as is necessary with specialized secure operating systems.
- They are flexible. Kernel hypervisors can be used both to implement a variety of different types of security policies and to provide replication functionality. Some of the possible applications are discussed below. A key feature is that kernel hypervisors can be "stacked", so that modular security policies can be developed and implemented as needed.
- They are not platform specific. Kernel hypervisors can be used on any operating system that supports kernel loadable modules that have access to the system call data structure. This includes Linux, Solaris, and other modern Unix systems, as well as Windows NT.
- They can be used to wrap COTS software without any modification to the software (including, e.g., relinking). Kernel hypervisors can monitor the actions of the COTS software and react in a manner that is transparent to the COTS application, other than it may be denied access to certain system resources according to the policy being enforced by the kernel hypervisor.

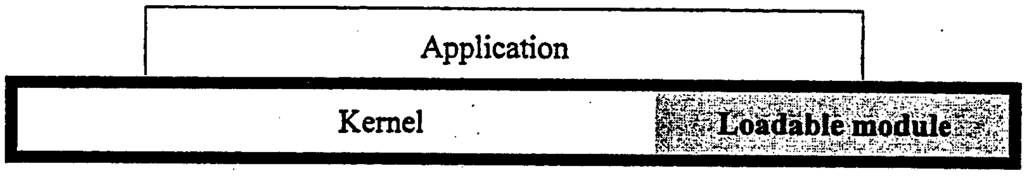
The goal of kernel hypervisors is to protect against malicious code or to provide some type of additional functionality. Hence, the operating assumption is that the user can be trusted. This assumption implies that kernel hypervisors are similar to virus protection programs in that a user must not specifically disable them. In fact, kernel hypervisors can be set up so that they cannot be disabled, but they do rely on proper user administration and use.

2.2 Components of the System

Using the concept of a kernel loadable module, we implemented kernel hypervisors on a Linux kernel. Linux was chosen as the initial platform because it supports loadable modules, the source code is free and easily available so the results of our work will be accessible to other researchers and developers, and it is widely used as a Web server platform and hence provides a good target for our approach.

As Figure 2 illustrates, Linux provides full support for kernel loadable modules[6]. Three system calls are supported that allow a privileged user to install and remove loadable modules and to list which modules are currently loaded. Once loaded, the kernel hypervisors in Linux run within kernel space and have full access to kernel data structures.

Figure 2 Linux Loadable Modules



Loadable modules operate within kernel space and have access to all kernel data structures.
Loadable modules can be loaded/unloaded without any modifications to the running system.
Commands:

lsmod	- list modules currently loaded
insmod *moduleName*	- load the module named *moduleName*
rmmod *moduleName*	- unload the module named *moduleName*

The remainder of this section discusses the three major components of our implemented kernel hypervisor system in more detail.

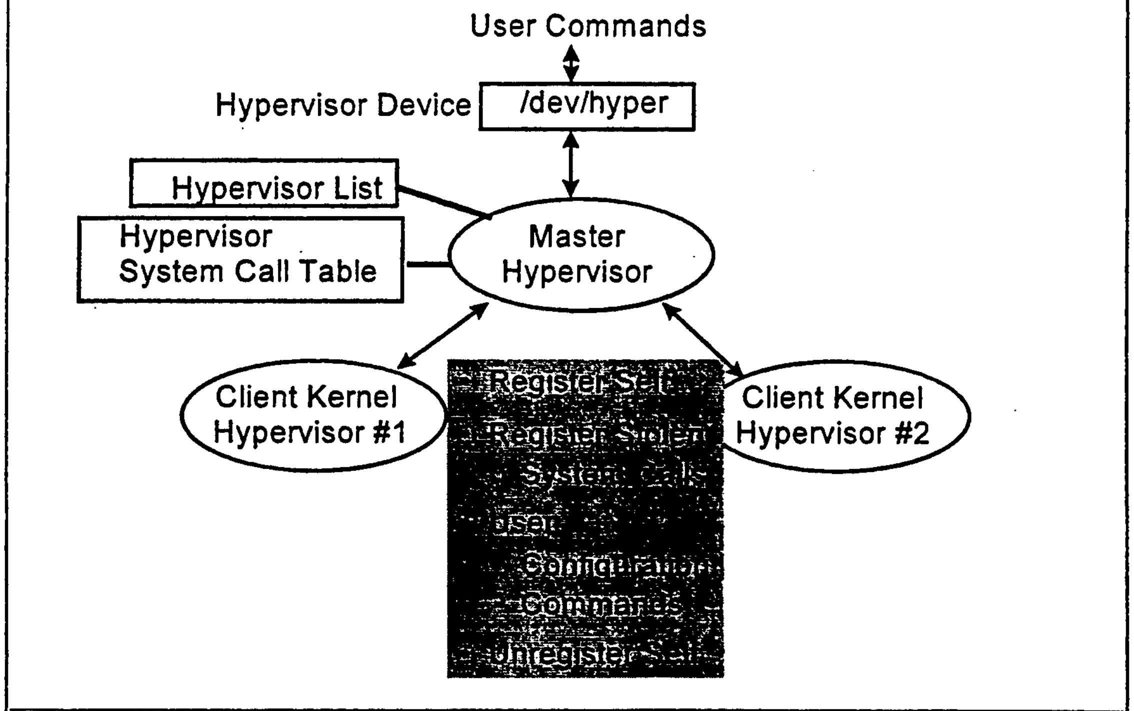
- The Master Hypervisor Framework
- The Client Kernel Hypervisors
- The Kernel Hypervisor Management.

2.2.1 The Master Hypervisor Framework

A framework has been developed based on a master kernel hypervisor, whose job is to coordinate installation and removal of individual client kernel hypervisors and to provide a means for management of these clients. The framework allows client kernel hypervisors to be stacked so that a variety of application specific policies can be implemented, each by means of its own kernel hypervisor. The hypervisors run in the kernel, but since they are loadable modules, they do not require that the kernel be modified.

Figure 3 illustrates the framework. The master hypervisor is loaded before any other client kernel hypervisors. A special application programming interface (API) has been defined that allows client hypervisors to register and unregister themselves with the master hypervisor and to identify which system calls they need to monitor. The master hypervisor keeps track of all currently registered client hypervisors and of the particular system calls that each client hypervisor is monitoring. When a client hypervisor module is removed via the rmmod call, it is the responsibility of the client hypervisor to de-register itself with the master hypervisor.

Figure 3. Master Hypervisor Functions

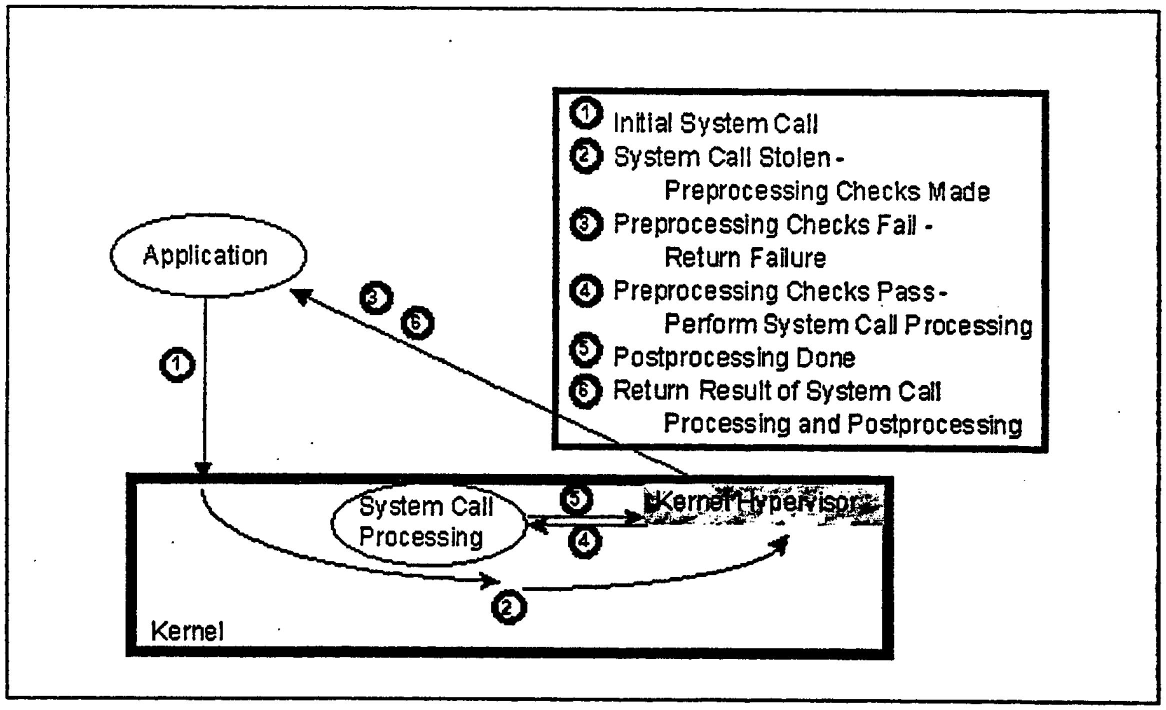


- Maintains hypervisor list
- Maintains hypervisor system call table
- Device driver for /dev/hyper

A special device, /dev/hyper, has been defined for communication between user space and kernel hypervisors. The master hypervisor acts as the device driver for this device. The device allows a privileged user to dynamically update the configuration information for a kernel hypervisor, including updating the security policy that the hypervisor enforces. It also provides a mechanism that kernel hypervisors can use to communicate with user space daemons. Such daemons, for example, could be used to provide additional audit capabilities or replication services.

The master hypervisor provides special wrapper code for use with any system call that is being monitored. The actual monitoring of system calls is performed by redirecting the links in the kernel system call table to point to system call wrappers that the master hypervisor provides. This redirection of links is the only modification to the kernel that is performed and is done by the master hypervisor only on system calls being monitored. The wrapper code is invoked when the system call is made and performs the processing illustrated in Figure 4. Each system call has its own

Figure 4. System Call Processing



1 Initial System Call
2 System Call Stolen - Preprocessing Checks Made
3 Preprocessing Checks Fail - Return Failure
4 Preprocessing Checks Pass - Perform System Call Processing
5 Postprocessing Done
6 Return Result of System Call Processing and Postprocessing
Application
System Call Processing
Kernel Hypervisor
Kernel

wrapper that follows the algorithm:

- For each client kernel hypervisor monitoring this call, initiate that client's pre-call processing.
- Call the standard system call processing.
- For each client kernel hypervisor monitoring this call, initiate that client's post-call processing.

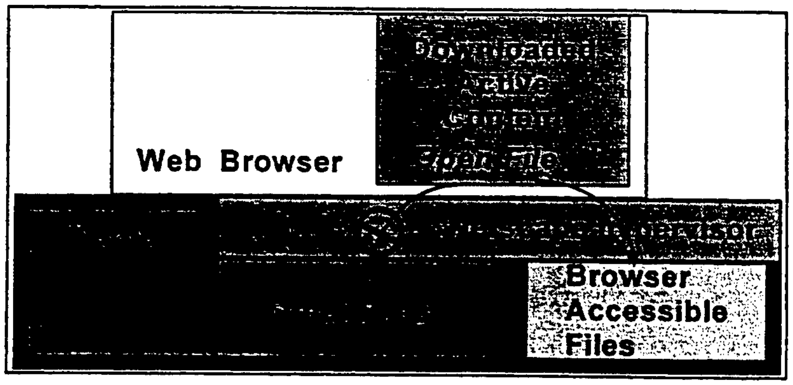
Pre-call and post-call processing is used to enforce the client hypervisor's particular security policy or to initiate other actions by the client hypervisor. This processing could include additional auditing of system calls, including call parameters and results; performing access checks and making access decisions for controlled resources that the hypervisor is protecting; modifying system call parameters; and passing information to user daemons.

2.2.2 Client Kernel Hypervisors

Client kernel hypervisors are developed and loaded separately as needed. A single client kernel hypervisor can be designed to monitor just one specific application or a number of different applications. Client kernel hypervisors could also be used to enforce other types of policies independent of any application.

To illustrate the practicality of the kernel hypervisor concept, we prototyped three client kernel hypervisors: one for wrapping the Netscape browser, one for replicating files, and one for wrapping the Apache Web Server.

2.2.2.1 Netscape Hypervisor



The goal of the Netscape hypervisor is to protect a user, browsing on the Internet, from downloading and executing malicious active content that might damage the user's system. The Netscape hypervisor accomplishes this by monitoring system calls made by the browser and enforcing a policy that only allows certain resources to be accessed. In particular, the set of files that the browser can open for read, and read/write access is controlled so that the browser effectively operates within its own limited execution context. While this does not prevent malicious code from accessing, and possibly damaging, resources within this context, it does limit the damage that could be done to only these resources.

In the case of the Netscape browser, the context includes the user's .netscape directory as well as limited access to other libraries needed by the browser to execute. Most files on the system, however, are not accessible to the browser and so cannot be damaged. To ensure that applications started from the browser as the result of a download, e.g. a postscript viewer, are also controlled, the hypervisor keeps track of all descendants of the browser and enforces the same policy on them as on the browser.

The security policy that the Netscape hypervisor enforces is stated as a set of rules identifying which resources the browser is allowed to access and what permissions the browser has to the resource. If there is no rule that allows access to a resource, then the hypervisor refuses any requests for access to that resource. The format of the rules is:

<type> *<identifier>* *<permissions>*

where *<type>* is either a file, socket, or process

<identifier> is either a file/dir pathname, an IP address, or a process ID

and *<permissions>* depends on the type.

For our Netscape prototype, only rules for the *<file>* type are used. For these rules permissions are:

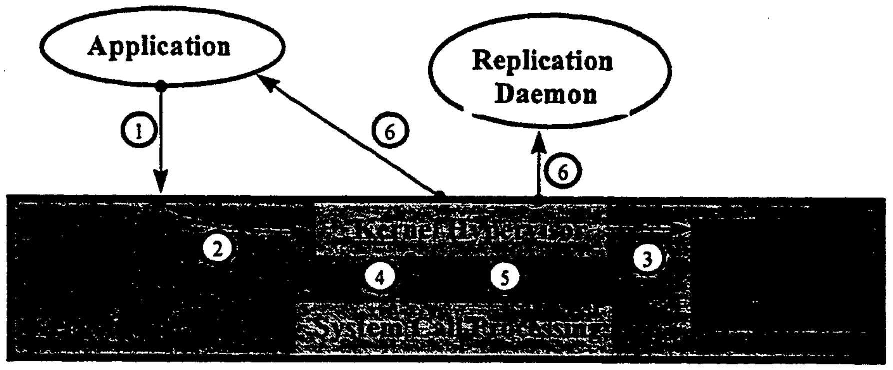
read, *write*, *read/write*, and *none*.

Certain conventions are used to simplify the statement of the rules. If an *<identifier>* is a file directory, then access to all files in that directory and all subdirectories is governed by the rule for the *<identifier>*, unless this rule is specifically overridden. Rules can be overridden by stating another, more specific, rule. For example, if a rule allows *read* access to all files and subdirectories of the directory /etc, then you can prevent users from accessing the file /etc/passwd by including a rule for this file with a permission of *none*.

2.2.2.2 Replication Hypervisor

The replication hypervisor is used to transparently replicate a file or set of files. The objective is to provide a replication facility that allows immediate backup of changes to a file without having to modify any applications that are making the actual changes.

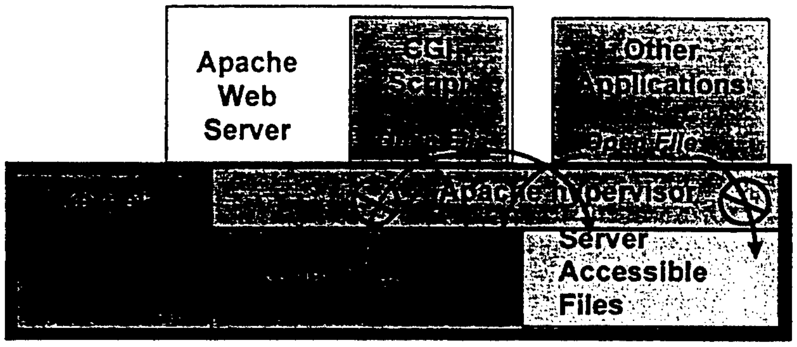
The replication hypervisor monitors all system calls that modify files looking for calls that modify the set of files to be replicated. When such a call is identified, the hypervisor caches the input parameters and allows the call to continue execution. If execution of the call completes successfully, then the hypervisor sends the cached input parameters to a replication daemon, operating in user space, that replays the call with the cached parameters on the copy of the file that it is maintaining.



Files can be replicated locally or across the network (using NFS) via this method. Results of our performance testing are documented in the Test Report, CDRL A006.

2.2.2.3 Apache Server Hypervisor

Based on our experience with the Netscape kernel hypervisor, a generic security hypervisor was developed that can be used to isolate any application in a compartment from which its access to files can be completely controlled. This generic security hypervisor, plus additional restrictions using techniques developed for the replication hypervisor, were used to develop a security hypervisor for the Apache Web Server.



The web server requires two types of protection. First, the server must be restricted to a subset of the file system so that any attack coming through the server will be contained. This was accomplished by creating a new hypervisor, hyper_apache. Hyper_apache provides for the apache server application, httpd, what hyper_ns does for netscape.

The server's temporary scratch directory and logging directories are configured as read/write while the html files presented by the server are configured as read only by hyper_apache.

The second piece of server protection is to isolate the configuration and html files used by the server from modification by un-authorized individuals. This is accomplished by a new hypervisor, hyper_cop (for Circle of Protection). Once installed and configured, hyper_cop restricts access to the files within its circle from all but certain privileged users. This will prevent a rouge user from overriding the system and changing the content of the web site even if they get access to root. The attacker will be required to know who the privileged user is before changes are allowed.

This implementation of hyper_cop is sufficient to show proof of concept but some open issues remain. These are discussed more fully in Section 5.

2.2.3 Kernel Hypervisor Management

The kernel hypervisor management component allows a user to obtain information on client hypervisors that are currently loaded and to reconfigure them, if needed. (As noted earlier, hypervisors are loaded and removed through standard Linux system calls.)

All hypervisor management is done via communication through the /dev/hyper device. The master hypervisor responds to requests for a list of all currently installed client hypervisors and their current identifiers. These identifiers are used to direct requests to specific kernel hypervisors.

The management functions available for the Netscape hypervisor include:

- the ability to list the current rules that are being enforced
- the ability to clear the current rule set and load a new one
- the ability to change the log level that the hypervisor uses to determine what detail of logging it should do.

2.3 Possible Applications

Kernel hypervisors can be used in a variety of ways to enhance the security and robustness of a system. This section discusses some of the types of uses followed by some specific applications.

2.3.1 Types of Uses

Auditing: In their simplest form, kernel hypervisors can provide an audit and monitoring functionality that merely records additional information about the system resources that an application is accessing. Such audit hypervisors are also useful for determining what system resources an application accesses during its normal processing. Since the level of auditing can be dynamically adjusted and since kernel hypervisors also provide a control mechanism, they could be useful as a component of an intrusion/detection/response system.

Fine-grained access control: The most compelling use of kernel hypervisors is to provide dynamic, fine-grained access control to various system resources such as files, network sockets and processes. The type of control possible is what differentiates this method of wrapping applications from standard access control list methods that perform control based on a user attribute. Kernel hypervisor security policies can be based on users, but can also be based directly on the application. For example, the Netscape kernel hypervisor that we implemented only monitored the Netscape application as well as any other applications started from within Netscape. Users had additional privileges to access their own .netscape directories, but could not override the restrictions in the Netscape hypervisor security database. These restrictions apply even to the root user. (In fact, an additional restriction imposed by the Netscape hypervisor was that the root user could not even run the Netscape browser.)

Label-based access control: While the rule sets described in Section 2.2 can be used to implement most simple policies, it would also be possible to use kernel hypervisors to implement more sophisticated label-based policies, such as a multilevel secure (MLS) policy or a type enforcement policy. To be able to implement these policies, a kernel hypervisor would need a way to label system resources. This might be done by creating and maintaining its own list of labeled names, or by piggybacking the labels on system data structures that have available space. Because of the flexibility of the kernel hypervisor, these label-based policies could be applied to only those portions of the system that required labels. And policies could be quickly changed, if needed, to adapt to the current operating environment.

Replication: Kernel hypervisors can also be used to provide replication features by duplicating system calls that modify system resources, such as files. We have investigated one approach for doing this by using a user daemon to replay file access requests that are intercepted by a kernel hypervisor. The replayed requests effectively duplicate that portion of the file system that is being monitored.

2.3.2 Specific Applications

Specific applications that kernel hypervisors can be used for include:

- Wrapping web browsers to protect the user from downloaded malicious active content.

  This is described in more detail in the section on the Netscape hypervisor. The most interesting point is that any applications that are started automatically from the browser, as well as any plug-ins, are also subject to the same restrictions that are on the browser. As noted earlier, this limits the damage that malicious active content can do to only those areas of the system that the user allows the browser to access.

- Wrapping web servers to protect against malicious attacks.

  Kernel hypervisors can be used to ensure that if the web server is overrun the damage is limited. In particular, it can ensure that files being served by the web server are not modified inappropriately. Also, because they are often easy to overrun, CGI scripts are sometimes attacked by malicious users on the web. In particular, if a CGI script has the ability to connect to an internal system, then an attacker might be able to compromise this CGI script to launch an attack. By limiting which ports on an internal system a CGI script can connect to, a kernel hypervisor can limit the attacker to only services on those ports.

- Wrapping services and proxies on an application gateway.

  Services, like Sendmail, often have unknown bugs that are only discovered when someone uses them to attack the system on which the service is running. Once again, kernel hypervisors provide a way of limiting the damage that such a compromise can cause.

- Adding new security features to a system.

  Kernel hypervisors can be used to add security features, such as security levels, roles, or domains and types, to a system without requiring any additional modifications to the system. In fact, a special kernel security hypervisor could be implemented that performs all of the required security checks for any other kernel hypervisor.

This is only a partial list of the possible security uses of kernel hypervisors. Any number of applications could benefit from the additional security that they can provide.

2.4 Summary

Kernel hypervisors are an approach to wrapping applications to provide additional security that has a number of advantages over other approaches. Because they reside in the kernel, they cannot be bypassed. Because they are implemented as loadable modules, they do not require any modification to the kernel. Because they do not require modification to an application, they can be used to dynamically wrap processes that are started from other processes that have already been wrapped. Because they can be easily configured, the policy that they enforce can be dynamically modified as needed. By limiting the amount of damage malicious software can do, kernel hypervisors provide an approach to protecting one's system against current and future threats that may still be unknown.

3. Performance Test Summary

Since kernel hypervisors perform additional processing on system calls that are being monitored, the testing goal was to measure how this additional processing affects the overall performance of the system and the performance of the particular applications that are being monitored. Performance testing was performed on two kernel hypervisors: the Netscape hypervisor and the replication hypervisor. This section summarizes the tests performed and the results of the testing. More detailed information can be found in the Test Report [7].

3.1 Netscape Kernel Hypervisor Tests

Since each process that is spawned by the Netscape browser is also monitored by the Netscape hypervisor, we used a shell process that could be spawned from within Netscape to run our tests. Each test consists of a C program that was run from within the shell. Times were measured for the following three situations:

1. The Netscape hypervisor is not running and the tests are run from within a shell. This case determines the baseline times that the other tests can be compared against.
2. The Netscape hypervisor is running and the shell is spawned from within Netscape. In this case we are measuring the full performance impact of the hypervisor since all monitored system calls made from within the shell will be checked.
3. The Netscape hypervisor is running and the shell is spawned from outside Netscape. In this case the Netscape hypervisor checks to determine if the process has been spawned by Netscape. When it determines that it was not, it does no further checking. In effect, this case measures the overhead that running the hypervisor has on the rest of the system.

3.2 Replication Kernel Hypervisor Tests

The replication task increases the processing on the monitored system calls: the path names and file descriptors are checked and saved, the writing buffers are duplicated, and communication messages are created and delivered.

The test results are affected largely by the length of file paths, the number of monitored file paths, the position of a file path in the check list, and the number of processes running on the system.

The tests repeatedly measured individual system calls and generated average time for executing the system calls. Times were measured for the following three situations:

1. The system calls access files when neither the replication hypervisor nor the replication daemon are running. This case determines the baseline times that the other tests can be compared against.
2. Both the replication hypervisor and the replication daemon are running, and the system calls access files that are being replicated. This case measures the full performance impact on system calls when the replication hypervisor performs all of the processing needed to replicate a file.

3. Both the replication hypervisor and the replication daemon are running, and the system calls access files that are not being replicated. This case measures the performance impact of merely checking whether a file is being replicated or not, but not doing any of the additional replication processing. In effect, this case measures the overhead that running the hypervisor has on the rest of the system.

3.3 Test Results

The performance testing for both hypervisors looked at two areas:

1. How did the hypervisor affect the performance when the hypervisor processing was not triggered by the system call? That is, how were the parts of the system that were not being controlled by the hypervisor affected.

   In this case the only additional processing was that performed by the hypervisor to determine whether it should invoke its special processing. For both the Netscape hypervisor and the replication hypervisor, this overhead was minimal, ranging from no measured overhead to a maximum overhead of 18 microseconds. This shows that hypervisors do not have an overly adverse performance impact on the rest of the system.

2. How did the hypervisor affect the performance of system calls, and hence applications, that were actually being controlled by the hypervisor?

   In this case there was a noticeable increase in the system call time. The increases ranged from 38-58 microseconds for the Netscape hypervisor and from 17-196 microseconds for the replication hypervisor. In many cases, the times for a system call being replicated by the replication hypervisor were close to twice the time of an unreplicated call. This was to be expected, since, as part of the replication, the hypervisor needed to make a copy of all incoming system call parameters.

It should be noted, that although there were measurable differences in system call response time when the hypervisors were running as opposed to when they were not, these differences were only of the magnitude of 100 microseconds or less and only for the system calls made by applications actually being monitored. In most cases, these differences will not be noticeable, and, at any rate, are certainly within the tolerable range, especially considering the functionality that the hypervisors provide. For example, we have not noticed any performance penalty from running the Netscape browser hypervisor.

4. Composition Analysis Summary

This section contains a short summary of the composition analysis that was done on the program. The full set of specifications and documentation of the composition analysis is in the Formal Specifications report [8].

Just as a software developer can build a complex system from a set of software modules, composition allows a systems analyst to build specifications for, and reason about, a complex system from the specifications of its components. While this suggests a "bottom up" approach to system specification, the "top down" approach, referred to as refinement, proves to be as useful if not more so. However, whether the analyst applies composition from the top down or from the bottom up, the analyst is able to reduce arguments about the properties of an entire, and possibly complex, system to a set of simpler properties and relatively simple arguments about each of the components.

In particular, the application of composition to the analysis of a system provides similar benefits as the modular approach to writing code. Even if the design (and specification) of one component changes, as long as the properties and interfaces between that component and the others remain stable, then only the arguments made about the properties of that one component need to be re-verified. In other words, the assurance arguments for unchanged components become reusable.

4.1 Specified Systems

The composition framework that was used is the one developed on the Composability for Secure Systems program which is the successor to the framework developed under the DTOS program. This framework is based on the work of Abadi and Lamport[9] and of Shankar[10]. The differences between the Composability and DTOS approaches and that of Abadi and Lamport are described in more detail in the Composability for Secure Systems Refined Design Report [11] and in the DTOS Composability Report[12] respectively. For both the Composability for Secure Systems program and the DTOS program, the framework has been specified in the PVS language.

The composability framework is based on a state transition model. Each *transition* is a triple of the form *(initial state, final state, transition agent)*. Depending on the needs of the analysis, the state may represent either the component state or the system state. The transition agent can represent a variety of concepts, including particular components, particular operations that a component supports, or threads that implement a component. Transition agents also can represent agents in the component's environment that are not part of the component.

Four separate, but related, systems were specified. The four specifications are related via refinement. The second and third specifications are refinements of the first; the fourth is a refinement of the second. The systems are:

- no hypervisor loaded into the kernel; that is, just one component: the kernel. Since this system has just a single component, composability has nothing to add to the analysis of it. However, the kernel without any hypervisors loaded into it forms the baseline system from which all of the other systems are built.

- a two component system consisting of the kernel and a single security hypervisor that monitors write requests loaded into the kernel
- an enhanced two component system consisting of the kernel and a single security hypervisor that monitors write requests and return parameters
- a multiple component system consisting of the kernel, a master hypervisor and a set of security hypervisors that monitor write requests. In this specification, the single kernel hypervisor has been refined into a master hypervisor and a set of hypervisors each of which have a security policy defined. It is possible to implement a single hypervisor that would behave identically to a system composed of the components specified in this case, but the added complexity of specifying the security policy would make the task very complex.

  In this model, all of the hypervisors are monitoring the same system call: a write request. The master hypervisor calls all hypervisors monitoring the given process before forwarding the write request to the kernel. In practice, the master hypervisor maintains a list of hypervisors monitoring a given system call. It calls each of the hypervisors monitoring that call, and each hypervisor in turn determines whether it monitors the given process.

Each specification describes the individual components of the system and the common state for the composite system. A list of properties satisfied by each component was derived, and composability theory was used to show that these properties are satisfied by the composite system.

4.2 Conclusions

Since composability theory says that the properties of a system are just the conjunction of the component properties, the analysis seems to be trivial. In fact, the hypotheses placed on a set of components for composability are so weak, that the composition theorem seems to say almost nothing. However, the final results are not as trivial as they appear at first glance, and this is often the case with any mathematical construct which is defined well. The strength of composition comes not from the composition theorem *per se*, but from the way in which composition of two components is defined.

The interesting features of a specification come in the decision of what should be specified in each component's transitions, and what other transitions each component can tolerate. In fact, much of the strength of the properties relies on what things one can reasonably enforce on the environment of a component. The more that one can enforce on the environment of a component, the more strongly one can state (and perhaps more easily one can prove) the properties of the component. However, the counter argument is that specifying too much in the environments of the system components can specify away all functionality of the composite system (since the composite consists of only those individual component transactions that satisfy each of the other component's environment transaction specifications). The goal therefore is to specify in each component's environment only those things that are needed to prove the component's properties hold.

The first specification represents an operating system that is not protected by a kernel hypervisor. The final three specifications represent refinements of a sort on that initial specification. They are not true refinements in the sense that the hypervisors are enforcing a security policy in addition to the one already enforced by the kernel. However, this appears to be the most powerful application of composition. While the code for the kernel itself may be unavailable to the developers of the hypervisors, the hypervisors are kernel-loadable modules, and thus represent a refinement of the kernel. Since the code for the kernel may be off limits, the interfaces and properties of the kernel are both well-defined and stable.

This is an ideal situation for the re-use of the specification of the kernel properties. We know what they are to begin with. After loading the hypervisors into the kernel, the properties of composite system are just the conjunction of the properties of the hypervisors and the kernel. That means that we can layer the desired behavior of the hypervisors over the properties of the kernel with the assurance that these properties will also hold.

5. Lessons Learned and Future Directions

This section contains a summary of lessons learned, open issues and possible future directions that the work described in this report could take.

5.1 Lessons Learned

An original goal of the program was to investigate whether kernel loadable modules could be used to add additional security and robustness to systems with minimal impact on the system and system applications. The results of the program clearly demonstrate that this is not only possible, but, in fact, has many advantages over other approaches for wrapping system components. These advantages include:

- No modifications are required to kernel code.
- No modifications are required to application code and monitored applications do not need to be recompiled or relinked.
- The wrapper code runs within the kernel and, hence, is much harder to bypass.
- Application policies can be easily changed or updated in a dynamic manner.
- Wrappers can be easily stacked to enforce more complex policies that consist of a number of simpler policies.
- The approach can be used on any system that supports kernel loadable modules.
- The approach can be used to implement a wide variety of security, integrity and redundancy features. Since access to specific system calls can be monitored, additional controls on what system calls a user can execute can be added.

One of the more interesting results of our work is the validation of the concept of a master kernel hypervisor to control the specific client hypervisors and to provide communication between user space and specific hypervisors for management. An alternative approach would be to construct one large loadable module that contained all of the functionality of both the master kernel hypervisor and the specific client hypervisors. This large module approach would be much less flexible, manageable and maintainable. Hence, proving that the more modular approach used on the program actually worked is a significant accomplishment.

Another interesting result is the development of a generic kernel hypervisor that can be used to provide an isolated environment within which an application can execute. As demonstrated with both the Netscape hypervisor and the Apache Web server hypervisor, the generic hypervisor is easy to configure for applications with minimal work. By running the hypervisor in audit mode, the files that the application normally accesses can be identified and the appropriate configuration file constructed that allows only the accesses needed by the application.

The hyper_cop hypervisor provides the other most common type of additional access control needed: controlling the accesses to particular files for all applications and users on the system. The combination of the generic hypervisor with the hyper_cop hypervisor provides most of the additional access control that might be desired.

There are still some open issues, however, involving the hyper_cop hypervisor. These are discussed in the next section.

5.2 Open Issues

While the feasibility and usability of kernel hypervisors was shown on the program, there are a number of issues that arose that remain open and will require additional research to resolve. The most interesting areas that were not fully addressed on the program due to limitations on time and funding include:

- Network and process control.
  While the kernel hypervisor framework is appropriate for controlling access to files, network resources and processes, only file access control was investigated on this program. It remains an open issue as to how well the approach will work for controlling communications between processes and for controlling access at the network level (for example, to limit access to specific IP addresses or to perform additional processing as part of a network request).

- Platform independence.
  While the kernel hypervisor approach should work for any system that supports kernel loadable modules, it has only been tested on the Linux operating system. For other Unix systems, like Solaris, that support kernel loadable modules, it should be a relatively straight forward task to port the current framework. For a system like Windows NT, that is proprietary with little kernel documentation, it remains an open issue as to how well the approach will work. There are unofficial ways to insert code to intercept system calls on NT that have been documented [13]. However, whether these methods are sufficient to support the hypervisor framework is unknown.

- Handling relative path names.
  The current hyper_cop prototype is sufficient to show proof of concept but is weak on protection. Hyper_cop is configured by reading in and storing a list of filenames to protect, which it checks each time an open() call is encountered. The filename being opened is checked against the list and access decision are made. Unfortunately, relying on filename string comparisons is a very inaccurate way to identify files. To circumvent the protection, a user simply needs to access the file by a different name than the name being protected by hyper_cop. This can be accomplished by many methods including accessing the file through a symbolic link, through the /proc directory structure or simply by giving a relative path and name instead of the full name of the file. How difficult it is to identify and control all such indirect accesses to a file is an open issue.

5.3 Future Directions

The kernel hypervisor concept is extremely flexible and has a variety of possible applications. This implies that there are a number of possible directions for future research and development in this area. These include:

- Extending the approach to other platforms.
  The first step in doing this is to adapt the technology, which currently runs only on the Linux operating system, to other popular operating systems, in particular Sun Solaris and Windows NT.
  The Solaris operating system supports a flexible, kernel loadable module functionality that is more than sufficient to implement the kernel hypervisor framework. In fact, it has operating system features similar to, and in some cases more sophisticated than, those provided by Linux and used on the Kernel Hypervisor program.
  While the Kernel Loadable Wrapper architecture for NT would be the same as for Linux and Solaris, because of the differences in the operating systems, the NT development would be more difficult. The primary difficulty arises from the fact that Windows NT is a closed, proprietary system and Microsoft does not release documentation of some of the kernel details that would be needed.

- Enhancing the capabilities of the hyper_cop hypervisor.
  Stronger protection would be accomplished by developing a shell that works with hyper_cop to provide user authentication. Such a hypervisor/shell combination would assure only the desired individuals would have access to the protected files regardless of root access or using trial and error to identify the privileged user.
  More work needs to be done to ensure that the hypervisor can correctly identify file references that are made via either relative pathnames or alternate paths.

- Developing additional types of hypervisors.
  The current set of hypervisors are all focused on file access, audit and replication. Other areas where hypervisors could be useful and should be developed include interprocess communication and network access and audit.

- Extending the composition analysis.
  The composition framework developed under the DTOS and Composability For Secure Systems programs was specified in PVS, which provides an interactive environment for writing specifications and machine checking formal proofs. Ideally, the specifications done on this program would have also been specified in PVS, and the properties stated for each component could have been proved by the theorem checking capability of PVS. However, time and budget constraints of this program led us to choose not to specify the components in PVS. As the specifications become more complex the additional effort to write specifications in PVS pays off in higher confidence in their correctness, and minor changes to the specifications allow the analysts to re-run machine proofs according to existing strategies rather than having to re-prove a theorem by hand.

- Protecting additional applications.
  There are a number of additional areas where specific client kernel hypervisors could be prototyped. These areas include:

- protecting security critical databases and servers
- ensuring that CORBA security controls are not bypassed
- enhancing audit capabilities dynamically
- providing dynamic file redo logs to support recoverability.

5.4 Conclusion

The kernel hypervisor program has successfully demonstrated the feasibility and usefulness of using kernel loadable modules to add additional security and robustness to an application without needing to modify either the application or the operating system on which the application runs. Future research should now focus on extending the framework developed on this program to other platforms and on developing new and enhanced client kernel hypervisors to provide security, reliability and recovery features not currently available.

6. References

[1] Thomas Bressoud and Fred Schneider. Hypervisor-based Fault-Tolerance. *Proceedings of the 15th ACM Symposium on Operating System Principles.* December, 1995. ACM Press.

[2] Terrence Mitchem, Raymond Lu, Richard O'Brien. Using Kernel Hypervisors to Secure Applications. *Proceedings of the 1997 Applied Computer Science Applications Conference.* December 1997.

[3] Secure Computing Corporation. Design Report, Hypervisors for Security and Robustness Program, CDRL A005. Feb 1998.

[4] M. Leech, et al. RFC 1928: SOCKS Protocol Version 5. March 1996.

[5] Ian Goldberg, David Wagner, Randi Thomas and Eric Brewer. A Secure Environment for Untrusted Helper Applications. *Proceedings of the 6th USENIX Security Symposium.* July, 1996.

[6] Bjorn Ekwall and Jacques Gelinas. Linux Modules 2.1.3 Documentation. June, 1996. Distributed with Linux source.

[7] Secure Computing Corporation. Test Results, Hypervisors for Security and Robustness Program, CDRL A006. Feb 1998.

[8] Secure Computing Corporation. Formal Specifications, Hypervisors for Security and Robustness Program, CDRL A007. Feb 1998.

[9] Martin Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems* 17, 3. May 1995.

[10]N Shankar. *A Lazy Approach to Compositional Verification.* Technical Report TSL-93-08, SRI International, December 1993.

[11]Secure Computing Corporation. Composability for Secure Systems Refined Design Report. February 6, 1998.

[12]Secure Computing Corporation. DTOS Composability Study. February 1997.

[13]Mark Russinovich and Bryce Cogswell. Windows NT System-Call Hooking. *Dr. Dobb's Journal,* January, 1997.

7. Appendix: User Guide for Kernel Hypervisors

User Guide Table of Contents

7.1 Overview

This document is the User Guide for the set of kernel hypervisors developed by Secure Computing Corporation to add security and robustness to the Linux operating system and certain COTS applications. A hypervisor is a layer of software, normally operating directly on a hardware platform, that implements the same instruction set as that hardware. They have traditionally been used to implement virtual machines.

Kernel hypervisors are a similar concept, but are implemented on top of an operating system kernel rather than on top of the hardware. These kernel hypervisors provide a set of "virtual" system calls for selected system components by intercepting system calls to perform pre-call and post-call processing. They can be used to make a component more robust or to perform various security functions, such as application specific fine-grained access control or auditing of kernel events.

Kernel hypervisors have a number of potential applications, including protecting user systems from malicious active content downloaded via a Web browser and wrapping servers and firewall services to limit possible compromises. To illustrate the possibilities, five hypervisors have been created for the Linux operating system; hyper_boss, hyper_ns, hyper_apache, hyper_cop and hyper_replicate.

Hyper_boss is the master hypervisor responsible for trapping system calls and re-directing them to its registered client hypervisors. Hyper_boss also serves as the device driver for /dev/hypers which allows communication between user space and kernel space.

Hyper_ns is a wrapper for the Netscape web browser. Hyper_ns restricts Netscape and any processes spawned by Netscape to a defined domain. This protects the user's system from attacks by downloaded active content. Any damage is restricted to the files and accesses listed in the hyper_ns domain.

Hyper_apache is a wrapper for the apache web server. Installing hyper_apache will restrict the file access of the apache web server to the defined domain and contain any attacks through the web server. In addition, by restricting the files which make up the content of the web site to Read Only, hyper_apache will protect the web content from modification by the server process or any process spawned by the server process. This may prevent attacks attempting to change the content of your web site.

Hyper_cop defines a circle of protection for one or more users. The files listed in a user's circle are restricted from the rest of the world so that only the circle owner is allowed the accesses defined for the circle files.

Hyper_replicate will replicate a defined file set to a second location. Once configured, the files to be replicated (target files) are copied to the destination. Then changes to the target files are tracked and duplicated in the destination set.

7.2 Installation Instructions and Brief Usage Notes.

a. Log in as root.

b. Make sure Linux is installed with loadable kernel modules enabled.

The Linux manual includes detailed instructions on configuring and installing a new kernel. If needed, follow the instructions for using "make config", "make

menuconfig" or "make xconfig" to configure a new kernel.
For a quick check if kernel modules are enabled, run the command "/sbin/lsmod" If modules are listed, you have kernel modules and should not have to reconfigure the kernel. If no modules are listed, you may have to re-configure.

c. Copy the hypervisor tar file to the /home directory and expand.
```
cp /mnt/floppy/hypers.gz /home
tar -cvzf hypers.gz
```
d. Create the hypervisor device to enable communication between kernel and user space.
```
mknod -m 666 /dev/hyper c 42 0
```
e. Copy default configuration files to /etc/hypervisor
```
cp /home/hypervisor/configs /etc/hypervisor
```
e. Change to the directory containing the compiled modules and configuration programs.
```
cd /home/hypervisor/bin
```
f. Install the master hypervisor.
```
insmod hyper_boss
```
g. Install desired client hypervisors. They will register themselves with the master hypervisor and be assigned a client number.
```
insmod hyper_ns
insmod hyper_apache
insmod hyper_cop
insmod hyper_replicate
```
h. Use vi, emacs, or your personal editor of choice to edit the configuration files. The default files include instructions and examples to help create configurations appropriate for your desired effect.

i. Use one of the client configuration applications to find out which client id was assigned to each hypervisor.
```
nsctl 0
copctl 0
repctl 0
```
Any one of these commands will cause the master hypervisor to print out configuration information including the assigned client ids. For example:
```
id = 1    name = hyper_netscape
          desc = hypervisor for netscape
          steal syscalls =    2    5    8    ...
id = 2    name = hyper_wrapper
          desc = generic application wrapper hypervisor
          steal syscalls =    ...
id = 3    name = hyper_cop
          ...etc...
```
j. Configure client hypervisors.
```
nsctl 1 1
nsctl 2 1 /etc/hypervisor/apache.conf
copctl 3 1
repctl 4 1
```
More detailed description on configuration of the client hypervisors can be found in the following sections. The highlights are:

1. nsctl is used for hyper_ns and hyper_apache but it assumes hyper_ns for default configuration. When nsctl is used for hyper_apache, the configuration file must be specified.

2. copctl is used for hyper_cop

3. repctl is used for hyper_replicate

4. the first parameter is the client id as reported by hyper_boss

5. usage information is listed when the control command is issued without any parameters. This also indicates the default configuration file.

7.3 Linux Commands

Three Linux commands are used to manage the hypervisors:

```
     /sbin/insmod    "module name"   -- install kernel module
     /sbin/rmmod     "module name"   -- remove kernel module
     /sbin/lsmod                     -- list modules
```

These commands must be executed by the super user.

Always install the master hypervisor "hyper_boss" before any of the client hypervisors. Trying to install a client hypervisor without hyper_boss present will result in error messages from the hyper_register, hyper_unregister and hyper_steal_syscall functions reporting

```
     wrong version or undefined
```

Always remove all client hypervisors before removing the master hypervisor. Trying to remove hyper_boss while another hypervisor is present will result in the error message

```
     Device or resource busy
```

7.4 hyper_boss

7.4.1 Description

A framework has been developed based on a master kernel hypervisor, hyper_boss, whose job is to coordinate installation and removal of individual client kernel hypervisors and to provide a means for management of these clients. The framework allows client kernel hypervisors to be stacked so that a variety of application specific or task specific policies can be implemented, each by means of its own kernel hypervisor. The hypervisors run in the kernel, but since they are loadable modules, they do not require that the kernel be modified.

The master hypervisor is loaded before any other client kernel hypervisors. A special application programming interface (API) has been defined that allows client hypervisors to register and unregister themselves with the master hypervisor and to identify which system calls they need to monitor. The master hypervisor keeps track of all currently registered client hypervisors and of the particular system calls that each client hypervisor is monitoring. When a client hypervisor module is removed via the rmmod call, it is the responsibility of the client hypervisor to de-register itself with the master hypervisor.

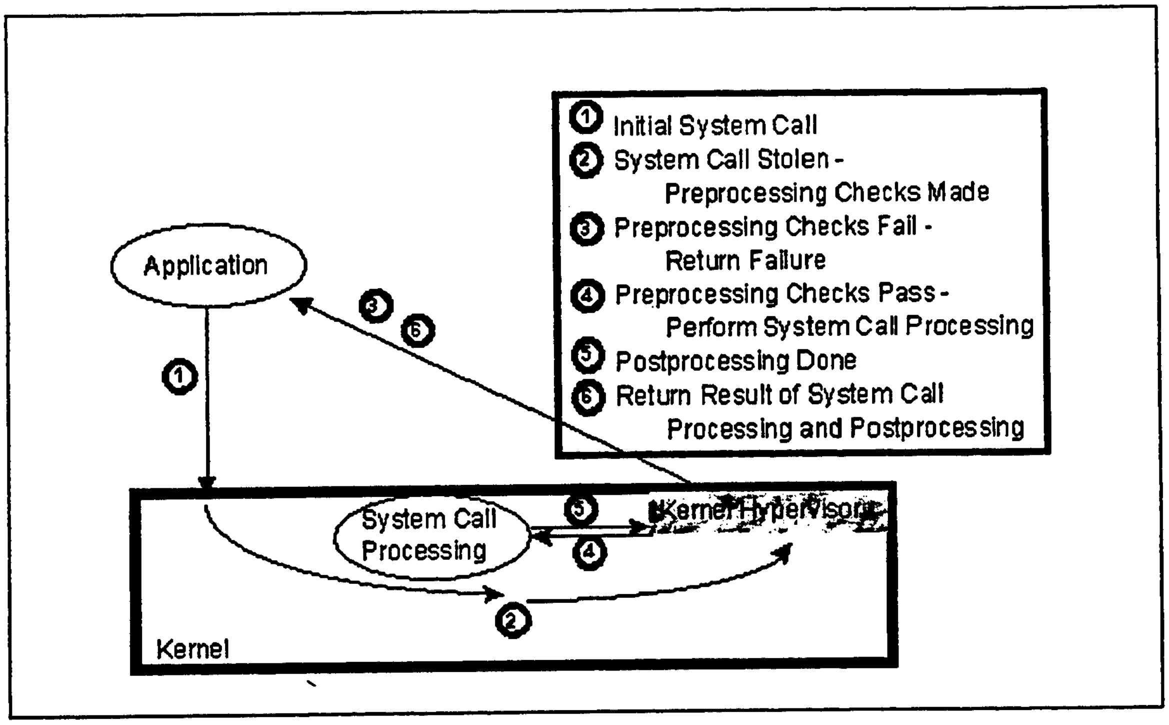
A special device, /dev/hyper, has been defined for communication between user space and kernel hypervisors. The master hypervisor acts as the device driver for this device. The device allows a privileged user to dynamically update the configuration information for a kernel hypervisor, including updating the security policy that the hypervisor enforces. It also provides a mechanism that kernel hypervisors can use to communicate with user space daemons. Such daemons, for example, could be used to provide additional audit capabilities or replication services.

The master hypervisor provides special wrapper code for use with any system call that is being monitored. The actual monitoring of system calls is performed by redirecting the links in the kernel system call table to point to system call wrappers that the master hypervisor provides. This redirection of links is the only modification to the kernel that is performed and is done by the master hypervisor only on system calls being monitored. The wrapper code is invoked when the system call is made and performs the processing illustrated in Figure 1.

Each system call has its own wrapper that follows the algorithm:

- For each client kernel hypervisor monitoring this call, initiate that client's pre-call processing.
- Call the standard system call processing.
- For each client kernel hypervisor monitoring this call, initiate that client's post-call processing.

Figure 1. System Call Processing


① Initial System Call
② System Call Stolen - Preprocessing Checks Made
③ Preprocessing Checks Fail - Return Failure
④ Preprocessing Checks Pass - Perform System Call Processing
⑤ Postprocessing Done
⑥ Return Result of System Call Processing and Postprocessing
Application
System Call Processing
Kernel Hypervisor
Kernel

Pre-call and post-call processing is used to enforce the client hypervisor's particular security policy or to initiate other actions by the client hypervisor. This processing could include additional auditing of system calls, including call parameters and results; performing access checks and making access decisions for controlled resources that the hypervisor is protecting; modifying system call parameters; and passing information to user daemons.

7.4.2 Usage

The following commands should be run from /home/hypervisor/bin.

/sbin/insmod hyper_boss	to load hyper_boss
/sbin/rmmod hyper_boss	to unload hyper_boss
/sbin/lsmod	to list modules loaded

Hyper_boss needs to be loaded before any of the other client hypervisors.
Hyper_boss can not be unloaded until after all client hypervisors have been unloaded.

Any of the hypervisor control applications (nsctl, copctl, repctl) with first parameter 0 (zero) can be used to get a report from hyper_boss of the registered client hypervisors, their client id, name, description and re-mapped system calls. For example, run:

```
/home/hypervisor/bin/nsctl 0
```

to get the client hypervisor report.

Messages are logged to the stdout terminal and to the file:
/var/log/messages

7.4.3 Configuration

The hypervisor device, /dev/hyper, must exist before nsctl, copctl or repctl can communicate to hyper_boss. This device is created by root running the command:

```
mknod -m 666 /dev/hyper c 42 0
```

7.4.4 Known Bugs/Limitations

None.

7.5 hyper_ns

7.5.1 Description

The goal of the Netscape hypervisor is to protect a user, browsing on the Internet, from downloading and executing malicious active content that might damage the user's system. The Netscape hypervisor accomplishes this by monitoring system calls made by the browser and enforcing a policy that only allows certain resources to be accessed. In particular, the set of files that the browser can open for read, and read/write access is controlled so that the browser effectively operates within its own limited execution context. While this does not prevent malicious code from accessing and possibly damaging resources within this context, it does limit the damage that could be done to only these resources.

In the case of the Netscape browser, the context includes the user's .netscape directory as well as limited access to other libraries needed by the browser to execute. Most files on the system, however, are not accessible to the browser and so cannot be damaged. To ensure that applications started from the browser as the result of a download, e.g. a postscript viewer, are also controlled, the hypervisor keeps track of all descendants of the browser and enforces the same policy on them as on the browser.

The security policy that the Netscape hypervisor enforces is stated as a set of rules identifying which resources the browser is allowed to access and what permissions the browser has to the resource. If there is no rule that allows access to a resource, then the hypervisor refuses any requests for access to that resource. The format of the rules is:

<type> *<identifier>* *<permissions>*

where *<type>* is either a file, socket, or process

<identifier> is either a file/dir pathname, an IP address, or a process ID

and *<permissions>* depends on the type.

For our Netscape prototype, only rules for the *<file>* type are used. For these rules permissions are:

read, *write*, *read/write*, and *none*.

Certain conventions are used to simplify the statement of the rules. If an *<identifier>* is a file directory, then access to all files in that directory and all subdirectories is governed by the rule for the *<identifier>*, unless this rule is specifically overridden. Rules can be overridden by stating another, more specific, rule. For example, if a rule allows *read* access to all files and subdirectories of the directory /etc, then you can prevent users from accessing the file /etc/passwd by including a rule for this file with a permission of *none*.

7.5.2 Usage

The following commands should be run from /home/hypervisor/bin.

```
/sbin/insmod hyper_ns          to load hyper_ns
/sbin/rmmod hyper_ns           to unload hyper_ns
/sbin/lsmod                    to list modules loaded
```

Hyper_boss needs to be loaded before hyper_ns.

Use the hypervisor client control application, nsctl, to control and configure hyper_ns. nsctl is located in /home/hypervisor/bin. Use:

```
nsctl                     to get a usage report showing how to use nsctl
nsctl 0                   to have hyper_boss list configured client hypervisors
nsctl [id] 0              to print hyper_ns configuration
     where [id] is the client id for hyper_ns as reported by hyper_boss in the
     configured client hypervisors report.
nsctl [id] 1              to configure hyper_ns from the default configuration file
     The default configuration file for nsctl is /etc/hypervisor/ns_acl.conf.
nsctl [id] 4              to clear all configuration
nsctl [id] 5 [log level]  to set the log level.
```

Messages are logged to the stdout terminal and to the file:

    /var/log/messages

7.5.3 Configuration

The configuration file, /etc/hypervisor/ns_acl.conf, is used with the command

    nsctl [id] 1

to configure hyper_ns.

An example configuration file,

    /home/hypervisor/configs/ns_acl.conf

is included with the distribution. This file's @general access control list shows the minimum accesses needed by netscape for the web browser to run correctly. The @target entry may need to be changed from /usr/local/bin/netscape to the correct path to the netscape application.

7.5.4 Known Bugs/Limitations

None.

7.6 *hyper_apache*

7.6.1 Description

Both hyper_ns and hyper_apache perform the same generic function; they wrap a specified application and restrict its access to a pre-defined domain. This concept can be applied to any application. In fact, changing the @target entry of the configuration file will allow either of our wrappers to watch over any application desired.

It may have been desirable to create one generic hypervisor, let's call it hyper_wrapper, which could be configured for whatever application needs to be wrapped. Then one could load hyper_wrapper as many times as needed to wrap all the desired applications. Unfortunately, the Linux system call *insmod* will not install a module more than once. So we are forced to have different names for the wrappers which handle Netscape and Apache.

Because of the *insmod* imposed limitation, and since we are specifically demonstrating wrappers for just the Netscape and Apache applications, hyper_apache was created to provide a second uniquely named hypervisor instead of creating a generic wrapper hypervisor.

The Apache Hypervisor, hyper_apache, started as an exact copy of hyper_ns. The only changes were to remove "netscape", "hyper_ns" and "ns" from all error messages and logging messages. These were changed to generic "wrapper" messages. Note that only messages were changed in making hyper_apache from hyper_ns. All the data structures, major components and minor functions are exactly the same.

Since hyper_apache is essentially hyper_ns with generic messages, one could easily create more wrapper hypervisors by the following steps.

- Copy hyper_apache.o to a new module name. For example, if the new hypervisor is to wrap the ftp server, it could be called something clever like hyper_ftp.o
- Create an appropriate configuration file with the desired target /usr/bin/ftp.
- Load and configure the new hypervisor just like hyper_ns or hyper_apache.

The only negative to having multiple hypervisors based on hyper_apache is that error messages and logging messages will not be unique.

7.6.2 Usage

The following commands should be run from /home/hypervisor/bin.

/sbin/insmod hyper_apache   to load

```
/sbin/rmmod hyper_apache    to unload
/sbin/lsmod                 to list modules loaded
```

Hyper_boss needs to be loaded before hyper_apache.

Since hyper_apache is just a more generic copy of hyper_ns, the same client control application, nsctl is used to control and configure hyper_apache or any other generic wrapper created from renaming hyper_apache. One just needs to make sure the correct configuration file is specified.

nsctl is located in /home/hypervisor/bin. Use:

```
nsctl                       to get a usage report showing how to use nsctl
nsctl 0                     to have hyper_boss list configured client hypervisors
```

hyper_apache and any other generic application wrapper hypervisor created by copying hyper_apache will be listed as "hyper_wrapper" by hyper_boss. If more than one wrapper is loaded, it is important to note the order they were loaded to match the client id with the desired hypervisor.

```
nsctl [id] 0                to print client hypervisor configuration.
```

where [id] is the client id for the wrapper hypervisor as reported by hyper_boss in the configured client hypervisors report.

```
nsctl [id] 1 /etc/hypervisor/apache.conf         to configure from the specified file.
```

the default configuration file for nsctl is /etc/hypervisor/ns_acl.conf so it is important to specify the configuration file for hyper_apache and any other generic wrapper hypervisors.

```
nsctl [id] 4                to clear all configuration
nsctl [id] 5 [log level]    to set the log level.
```

Messages are logged to the stdout terminal and to the file:
/var/log/messages

7.6.3 Configuration

The configuration file, /etc/hypervisor/apache.conf, is used with the command

```
nsctl [id] 1 /etc/hypervisor/apache.conf
```

to configure hyper_apache.

An example configuration file,
/home/hypervisor/configs/apache.conf
is included with the distribution. This file's @general access control list shows the minimum accesses needed by apache for the web server to run correctly. The @target entry may need to be changed from /usr/sbin/httpd if your system has httpd in a different location.

7.6.4 Known Bugs/Limitations

hyper_apache must be running before the apache server daemon, httpd is started. Since httpd is normally started up by the script S85httpd in /etc/rc.d/rc3.d before login is enabled, the user has three options.

- One, create a script, say S84hypers, to install hyper_boss and hyper_apache before S85httpd is started.
- Two, stop httpd with the command /etc/rc.d/rc3.d/S85httpd stop, start the hypervisors, then restart httpd with /etc/rc.d/rc3.d/S85httpd start.
- Three, remove S85httpd from the normal startup and run it manually after hyper_apache is installed.

In any case, for hyper_apache to trap the httpd process, you must make sure it is started before the first httpd is installed.

7.7 hyper_cop

7.7.1 Description

The Circle of Protection hypervisor, or hyper_cop, is a loadable kernel module and a client of the master hypervisor. The goal of hyper_cop is to restrict access to files which are listed as belonging to a specific user's Circle of Protection.

Much like hyper_ns, hyper_cop reads in an access control list which specifies files and accesses. However, with hyper_cop, there is no general access control list or a target application. There is no target application because hyper_cop does not protect files against access by a particular process but it controls all access to its protected files regardless of who or what application is requesting the access. There is no general access control list because adding a file to a circle is telling hyper_cop that the defined access is reserved for a specified user, so a user must be supplied. In fact, each listed file must belong to one and only one user's circle of protection.

The access associated with a file in a user's circle is interpreted as the access reserved for the circle owner. For example, an access of W (for write) specifies the owner of the circle reserves Write access to the file. All other users are allowed to Read but not Write. Reserving RW removes all Read and Write from general users and only the circle's owner is allowed any access.

In this way, a set of files can be restricted from general access while still being allowed access to a specified privileged user. This is more protective than the standard UNIX file control since even "root" will not be allowed access to protected files. To further protect the files from the super user, hyper_cop also watches for the Set User Id, (setuid) command and does not allow switching to a user who has a circle of protection configured. So even with root access, the super user will have to logout, log back in as the circle's owner and provide the correct password before access to the files is allowed. Eliminating setuid for circle owners forces users to authenticate themselves before being able to access circle files.

Note: A malicious user with root access could still change the password for a circle owner, then log in with the new password to gain access. To prevent this, a special user could add /etc/passwd to their circle with Write access reserved. Then root would no longer be able to change passwords.

7.7.2 Usage

The following commands should be run from /home/hypervisor/bin.

```
/sbin/insmod hyper_cop       to load
/sbin/rmmod hyper_cop        to unload
/sbin/lsmod                  to list modules loaded
```

Hyper_boss needs to be loaded before hyper_cop.

copctl is located in /home/hypervisor/bin. Use:

```
copctl                       to get a usage report showing how to use copctl
copctl 0                     to have hyper_boss list configured client hypervisors
copctl [id] 0                to print circle of protection hypervisor configuration.
        where [id] is the client id for the hyper_cop as reported by hyper_boss in the
        configured client hypervisors report.
copctl [id] 1                to configure hyper_cop from the default config file.
        The default configuration file for copctl is /etc/hypervisor/cop_acl.conf.
copctl [id] 4                to clear all configuration
copctl [id] 5 [log level]    to set the log level.
```

Messages are logged to the stdout terminal and to the file:

/var/log/messages

7.7.3 Configuration

The configuration file, /etc/hypervisor/cop_acl.conf, is used with the command

```
copctl [id] 1
```

to configure hyper_cop.

An example configuration file,

/home/hypervisor/configs/cop_acl.conf

is included with the distribution. This file shows how hyper_cop may be used to control access to the apache web server content. All users are still allowed Read access to the web content files but only a special user (the web master) is allowed to Write the files. You must change the @user name to an appropriate user or create a new user to match the one listed in cop_acl.conf.

Protecting the web content is just one of may uses for hyper_cop. Any set of files can be configured as belonging to a specified user's circle of protection. These files will then be protected from access by all other users even the super user.

Multiple circles can be configured for multiple users. Just remember to include a file in at most one user's circle for proper processing.

7.7.4 Known Bugs/Limitations

a. This implementation of hyper_cop is sufficient to show proof of concept but it is a bit weak on protection. Hyper_cop is configured by reading in and storing a list of filenames to protect, which it checks each time an open() call is encountered. The filename being opened is checked against the list and an access decision is made. Unfortunately, relying on filename string comparisons is a very inaccurate way to identify files. To circumvent the protection, a user simply needs to access the file by a different name than the name being protected by hyper_cop. This can be accomplished by many methods including accessing the file through a symbolic link, through the /proc directory structure or simply by giving a relative path and name instead of the full name of the file. For example, while /home/httpd/index.html may be known to hyper_cop as a file to protect, open() calls on /home/httpd/linktoindex.html, /proc/144/cwd/index.html, or ./index.html can not immediately be identified as the file to protect.

b. A file can exist in at most one user's circle. This is a rule assumed by the access logic of hyper_cop but is not enforced during configuration. If a file is listed in more that one user's circle, access control will not be properly enforced.

c. Files within a protected directory can still be deleted with "rm".

7.8 hyper_replicate

7.8.1 Description

The replication hypervisor is used to transparently replicate a file or set of files. The objective is to provide a replication facility that allows immediate backup of changes to a file without having to modify any applications that are making the actual changes.

The replication hypervisor monitors all system calls that modify files. It looks for calls that modify any of the files being replicated. When such a call is identified, the hypervisor caches the input parameters and allows the call to continue execution. If execution of the call completes successfully, then the hypervisor sends the cached input parameters to a replication daemon, operating in user space, that replays the call with the cached parameters on the copy of the file that it is maintaining.

Files can be replicated via this method either locally or across the network (using NFS).

7.8.2 Usage

The following commands should be run from /home/hypervisor/bin.

```
/sbin/insmod hyper_replicate  to load
/sbin/rmmod hyper_replicate   to unload
/sbin/lsmod                   to list modules loaded
```

Hyper_boss needs to be loaded before hyper_replicate.

repctl is located in /home/hypervisor/bin. Use:

repctl	to get a usage report showing how to use repctl
repctl 0	to have hyper_boss list configured client hypervisors
repctl [id] 0	to print replication hypervisor configuration.
	where [id] is the client id for hyper_replicate as reported by hyper_boss in the configured client hypervisors report. Note that a copy of repctl needs to be running in active replication mode for this command to show anything but "empty" for the monitored files.
repctl [id] 1	to configure hyper_replicate from the default config file. start replication and install the replicate deamon to service messages for modification of the replicated files.
	The default configuration file for repctl is /etc/hypervisor/replicate.conf.
repctl [id] 1 &	to config and run the replicate deamon in the background.
	The default configuration file for repctl is /etc/hypervisor/replicate.conf. Note the process id when running in the background so you can kill it if desired.
kill [pid]	to remove the replication deamon if running in background.
repctl [id] [debug level]	to set the debug level

Messages are logged to the stdout terminal and to the files:

/var/log/messages
/var/log/hypervisor/replicate

Note: the directory /var/log/hypervisor needs to be created with writing allowed before repctl [id] 1 is run.

7.8.3 Configuration

The configuration file, /etc/hypervisor/replicate.conf, is used with the command

```
repctl [id] 1 &
```

to configure hyper_replicate.

An example configuration file,

/home/hypervisor/configs/replicate.conf

is included with the distribution.

7.8.4 Known Bugs/Limitations

a. As with hyper_cop, hyper_replicate does not get the full pathname when a file is opened. Thus when a file is accessed by an application without using the full path, hyper_replicate does not recognize the file as one to replicate and replication will fail.

b. Since hyper_boss opens /var/log/messages when loaded and since hyper_boss is loaded before hyper_replicate, hyper_replicate will never trap the opening of /var/log/messages and will not replicate that file correctly.

7.9 Important Files and Directories

The following files and directories are used by the hypervisors for configuration files, log files, binary executables, source, etc.

- dir: /home/hypervisor

  Default home directory for the hypervisor source and executable files.

- dir: /home/hypervisor/bin

  Default location of the hypervisor modules and hypervisor client manager applications. The modules found here are:

hyper_boss.o:	master hypervisor
hyper_ns.o:	netscape hypervisor
hyper_apache.o:	generic hypervisor used to wrap the Apache server
hyper_cop.o:	Circle of Protection hypervisor.
hyper_replicate.o:	replication hypervisor.

  The client manager applications found here are:

nsctl:	to control hyper_ns and hyper_apache.
copctl:	to control hyper_cop.
repctl:	to control hyper_replicate.

- dir: /home/hypervisor/modules

  Default location of the hypervisor source code. Includes all source for all hypervisors and client manager applications.

- dir: /etc/hypervisor/

  Default location of the configuration files. The user may specify a new file name and location while configuring a hypervisor, but each client manager application has a default file which is loaded unless specifically changed. The default files are:

ns_acl.conf:	Default configuration file for hyper_ns and nsctl.
apache.conf:	Default configuration file for hyper_apache. Since nsctl is used to configure hyper_apache, and since the default file for nsctl is "ns_acl.conf", the user must specify /etc/hypervisor/apache_acl.conf when configuring hyper_apache.
cop_acl.conf:	Default configuration file for hyper_cop and copctl.
replicate.conf:	Default configuration file for hyper_replicate and repctl.

- file: /var/log/messages

Logging file for hyper_boss, hyper_ns, hyper_apache and hyper_cop.

- file: /var/log/hypervisor/replicate

  Logging file for hyper_replicate and the repctl deamon.

- file: /dev/hyper

  The device used for communication between user space and the kernel hypervisors. This device must exist to allow nsctl, copctl and repctl to talk to their hypervisors in kernel space. This device is created with the following command run by root: `mknod -m 666 /dev/hyper c 42 0`

DISTRIBUTION LIST

addresses	number of copies
AFRL/IFGB DWAYNE P. ALLAIN 525 BROOKS RD ROME NY 13441	3
SECURE COMPUTING CORP 2675 LONG LAKE RD ROSEVILLE, MN 55113	1
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
RELIABILITY ANALYSIS CENTER 201 MILL ST. ROME NY 13440-8200	1
ATTN: GWEN NGUYEN GIDEP P.O. BOX 8000 CORONA CA 91718-8000	1
AFIT ACADEMIC LIBRARY/LDEE 2950 P STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1

ATTN: TECHNICAL DOCUMENTS CENTER 1
OL AL HSC/HRG
2698 G STREET
WRIGHT-PATTERSON AFB OH 45433-7604

US ARMY SSDC 1
P.O. BOX 1500
ATTN: CSSD-IM-PA
HUNTSVILLE AL 35807-3801

NAVAL AIR WARFARE CENTER 1
WEAPONS DIVISION
CODE 48L000D
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

SPACE & NAVAL WARFARE SYSTEMS CMD 2
ATTN: PMW163-1 (R. SKIANO)RM 1044A
53560 HULL ST.
SAN DIEGO, CA 92152-5002

COMMANDER, SPACE & NAVAL WARFARE 1
SYSTEMS COMMAND (CODE 32)
2451 CRYSTAL DRIVE
ARLINGTON VA 22245-5200

CDR, US ARMY MISSILE COMMAND 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSMI-RD-CS-R, DOCS
REDSTONE ARSENAL AL 35898-5241

ADVISORY GROUP ON ELECTRON DEVICES 1
SUITE 500
1745 JEFFERSON DAVIS HIGHWAY
ARLINGTON VA 22202

REPORT COLLECTION, CIC-14 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

AEDC LIBRARY 1
TECHNICAL REPORTS FILE
100 KINDEL DRIVE, SUITE C211
ARNOLD AFB TN 37389-3211

COMMANDER 1
USAISC
ASHC-IMD-L, BLDG 61801
FT HUACHUCA AZ 85613-5000

US DEPT OF TRANSPORTATION LIBRARY 1
FB10A, M-457, RM 930
800 INDEPENDENCE AVE, SW
WASH DC 22591

AIR FORCE WEATHER TECHNICAL LIBRARY 1
151 PATTON AVE. RM 120
ASHEVILLE NC 28801-5002

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

SOFTWARE ENGINEERING INSTITUTE 1
CARNEGIE MELLON UNIVERSITY
4500 FIFTH AVENUE
PITTSBURGH PA 15213

NSA/CSS 1
K1
FT MEADE MD 20755-6000

ATTN: OM CHAUHAN 1
DCMC WICHITA
271 WEST THIRD STREET NORTH
SUITE 6000
WICHITA KS 67202-1212

AFRL/VSOS-TL (LIBRARY) 1
5 WRIGHT STREET
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

2

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.